A Requirements-Based Programming Approach to Developing a NASA Autonomous Ground Control System

James L. Rash (james.l.rash@nasa.gov)
NASA Goddard Space Flight Center

Michael G. Hinchey (michael.g.hinchey@nasa.gov)
NASA Goddard Space Flight Center

Christopher A. Rouff (rouffc@saic.com)
SAIC

Denis Gračanin (gracanin@vt.edu)
Virginia Tech

John Erickson (jderick@cs.utexas.edu)
University of Texas at Austin

Abstract. A new requirements-based programming approach to the engineering of computer-based systems offers not only an underlying formalism, but also full formal development from requirements capture through to the automatic generation of provably-correct code. The method, Requirements-to-Design-to-Code (R2D2C), is directly applicable to the development of autonomous systems and systems having autonomic properties. We describe both the R2D2C method and a prototype tool that embodies the method, and illustrate the applicability of the method by describing how the prototype tool could be used in the development of LOGOS, a NASA autonomous ground control system that exhibits autonomic behavior. Finally, we briefly discuss other possible areas of application of the approach.

Keywords: Validation, verification, formal methods, automatic code generation, NASA, autonomous systems, autonomic systems

1. Introduction

It has been argued that computer-based systems should be autonomic (Sterritt and Hinchey, 2005), and, more specifically, that autonomous systems are necessarily autonomic (Truszkowski et al., 2006). Further, it can be argued that autonomic systems are inherently autonomous, as they are required to adapt and evolve to meet their goals of being self-healing, self-configuring, self-optimizing, and self-protecting.

Autonomous systems can be exceedingly complex, and consequently extremely difficult to develop. Often, the complete behavior of the system cannot be foreseen at the outset, partly because of the evolving nature of the system, and partly because it is difficult to capture all of the necessary domain knowledge before development begins. Because of this and the system's emergent behavior (that is, behavior that is exhibited by a system as it evolves, but which was not anticipated) the system cannot be fully tested using traditional methods (Rouff et al., 2004).

Successes reported from the use of formal methods (Hinchey and Bowen, 1999) suggest they can go a long way towards solving these problems, and that they can reduce reliance on testing. However, they are still perceived to be difficult to use (Bowen and Hinchey, 1995), and their uptake in industry has not been as commonplace as one would have expected.

2. Requirements-Based Programming

2.1. Background

Requirements-Based Programming (RBP) has been advocated (Harel, 2001; Harel, 2004) as a viable means of developing complex, evolving systems. It embodies the idea that requirements can be systematically and mechanically transformed into executable code.

Generating code directly from requirements would enable software development to better accommodate the ever increasing demands on systems. In addition to increased software development productivity through eliminating manual efforts in the coding phase of the software lifecycle, RBP can also increase the quality of generated systems by automatically performing verification on the software—if the transformation is based on the formal foundations of computing.

This may seem to be an obvious goal in the engineering of software systems, but RBP does in fact go a step further than current development methods. System development typically assumes the existence of a model of reality, called a design (or, more correctly, a design specification), from which an implementation will be derived. This model must itself be derived from the system requirements, but there is a large "gap", termed the "analysis-specification gap," in going from requirements to design (Hinchey et al., 2005a)—representing the problem of capturing requirements and adequately representing them in a specification that is clear, concise, and complete. RBP seeks to eliminate this "gap" by ensuring that the ultimate implementation can be fully traced back to the actual requirements of the system (although, as usually proposed by its advocates, it does not necessarily entail full mathematical provability of the equivalence between a set of requirements and its implementation).

2.2. R2D2C

R2D2C (Requirements-to-Design-to-Code) is a NASA patent-pending approach to the engineering of complex computer systems where the need for correctness of the system, with respect to its requirements, is particularly high. This category includes NASA mission software, most of which exhibits both autonomous and autonomic properties.

The approach, described in greater detail in (Hinchey et al., 2005a), embodies the main idea of requirements-based programming. It goes further, however, in that the approach offers not only an underlying formalism, but also full formal development from requirements capture through to automatic generation of provably correct code. Moreover, the approach can be adapted to generate instructions in formats other than conventional programming languages—for example, instructions for controlling a physical device, or rules embodying the knowledge contained in an expert systemm. In these contexts, NASA is currently applying the approach to the verification of the instructions and procedures to be generated by the Hubble Space Telescope Robotic Servicing Missions (HRSM) and in the validation of the rule base used in the ground control of the ACE spacecraft.

In the remainder of this paper we describe a prototype tool to support the R2D2C method and report on our experiences in applying it to validate the prototype Lights-Out Ground Operations System (LOGOS), an autonomous system exhibiting autonomic properties (Truszkowski et al., 2006; Truszkowski et al., 2004; Rash et al., 2005).

3. Requirements to Design to Code

R2D2C takes, as input, system requirements written by engineers (and others) as scenarios in natural language, or UML use cases, or some other appropriate graphical or textual representation. From the scenarios, an automated theorem prover in which the laws of concurrency (Hinchey and Jarvis, 1995) have been embedded infers a corresponding process-based specification expressed in an appropriate formal language (currently we are using CSP, Hoare's language of Communicating Sequential Processes (Hoare, 1978; Hoare, 1985)).

For applications requiring the specification of time, such as real-time systems, a timed process-based specification language (e.g., timed CSP (Schneider et al., 1991)) could be used. Our current prototyping work addresses applications that do not require the specification of time.

A process-based specification is amenable to analysis and forms an appropriate basis for code generation. As much as possible, R2D2C

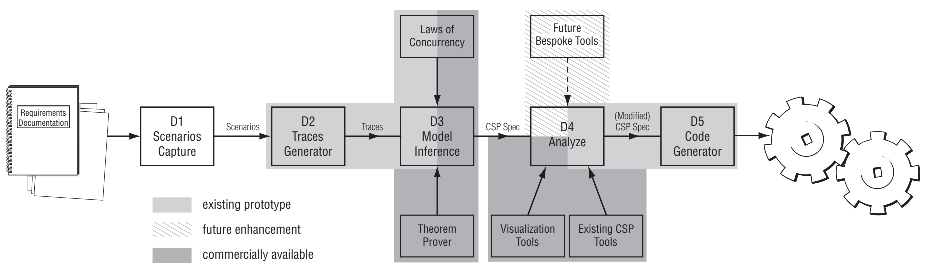


Figure 1. The R2D2C approach and current status of the prototype.

makes use of widely-available tools and notations that are well-trusted and that have been demonstrated to be useful in the development of high-quality systems.

A "short-cut" approach to R2D2C (Hinchey et al., 2004; Hinchey et al., 2005a) avoids the use of an automated theorem prover, which is computationally expensive. This alternative approach involves inferring a corresponding process-based specification (in a language we have named EzyCSP) without a theorem prover, but requires a (one time) proof of the translation in order to preserve the mathematical underpinnings of the R2D2C approach. Figure 1 illustrates those parts of the approach that are embodied in the current R2D2C prototype tool (described in the remainder of this paper), and shows where commercially-available and public domain tools may be used to support the approach.

3.1. Prototype Tool

The CSP formal model is the central part of the proposed approach, which conforms to a Model Driven Architecture (MDA) (Kleppe et al., 2003). The prototype tool automatically generates the code from the CSP model (or design) (Figure 2) into which the tool has already transformed the requirements.

Developing a tool based on CSP requires two major issues to be addressed—how to translate the CSP model into code and how to translate the requirements into the CSP model. The tool transforms the derived design (CSP model) into an equivalent software representation (code) using Java as the target programming language. There were several reasons for selecting the Java programming language (Gosling et al., 2000) both for tool implementation and for the target platform:

- Java is a general-purpose, concurrent, class-based, object-oriented programming language, with very few implementation and hardware dependencies.

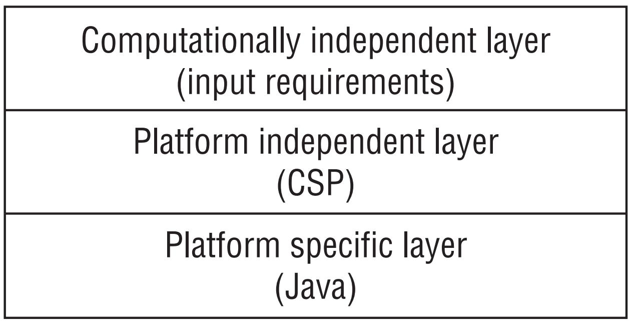


Figure 2. MDA approach

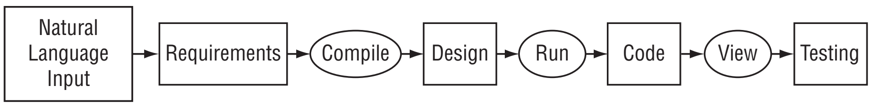


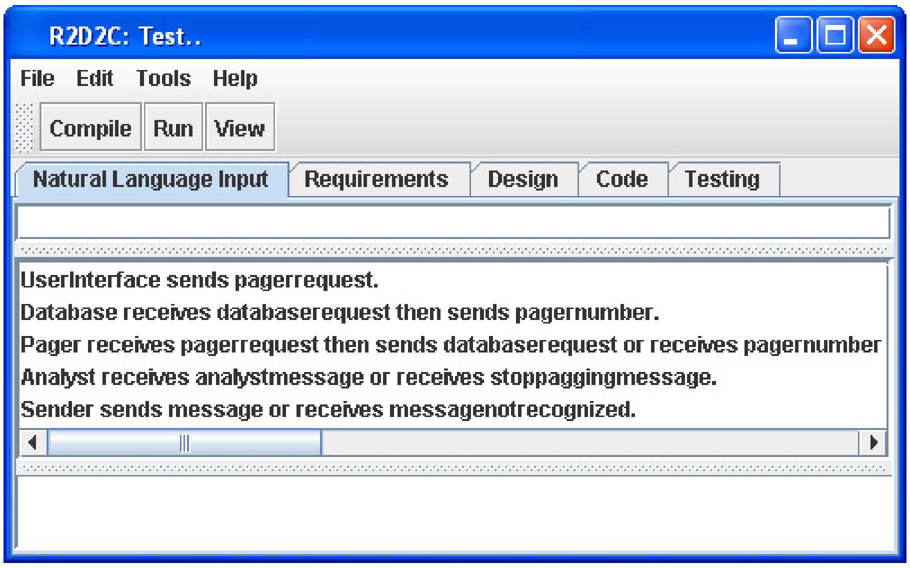
Figure 3. High-level program flow

- An off-the-shelf implementation (library) of CSP for Java[1] is available. While JCSP does not provide direct CSP-to-Java mapping, it conforms to the CSP model of communicating systems for Java multi-threaded applications (Lea, 2000). There is also support for distributed JCSP components using JCSP.net (Welch et al., 2002).
- Java Swing (Walrath et al., 2004), in combination with some available Java IDEs, greatly simplifies user interface development.
- Many Java-based translator development tools are available.

The prototype tool implementation in Java uses off-the-shelf components. A Swing-based user interface provides a transparent layer for entering the requirements and viewing the resulting model. Figure 3 shows the high-level program flow.

The translators are implemented using the ANTLR (Parr and Quong, 1995) tool, which provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions[2]. A discussion of ANTLR and some related tools can be found in (Smaragdakis et al., 2004). An English-like input language, specified as an ANTLR grammar, is used to specify user requirements (Figure 4). ANTLR uses the grammar to automatically generate the translator. The translator is then used to generate the CSP model that corresponds to the user requirements (Figure 5). Figure 6 shows the graph-based representation of the system (under development) (Rash et al., 2005).

[1] See http://www.cs.kent.ac.uk/projects/ofa/jcsp/
[2] See http://www.antlr.org/



UserInterface sends pagerrequest.
Database receives databaserequest then sends pagernumber.
Pager receives pagerrequest then sends databaserequest or receives pagernumber
Analyst receives analystmessage or receives stoppaggingmessage.
Sender sends message or receives messagenotrecognized.

Figure 4. Input requirements

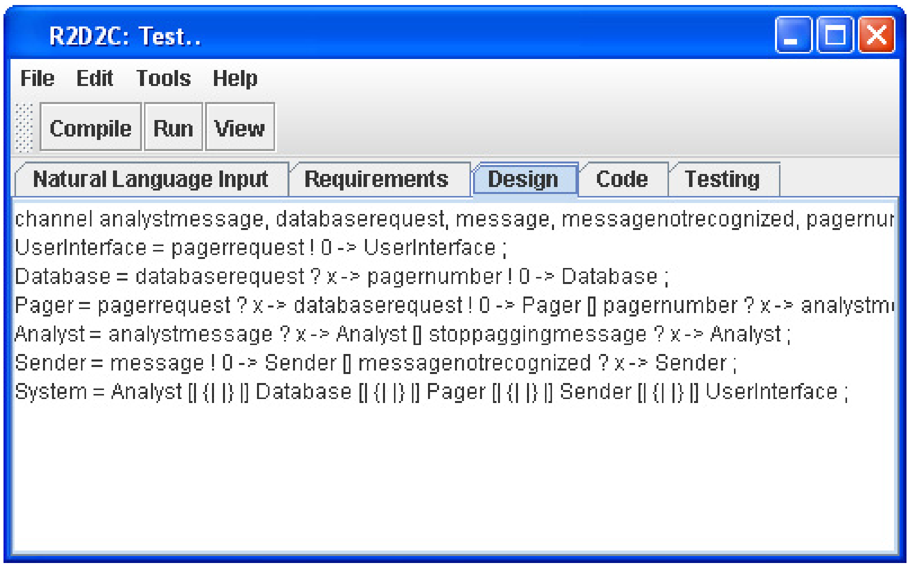
4. Experiences in Applying the Prototype Tool

4.1. LOGOS

The Lights-Out Ground Operations System (LOGOS) is a proof-of-concept NASA system for automating ground station operations in controlling satellites as they orbit the earth and periodically come into view of the ground station. Its design concepts take into account the need for adaptability to reflect variations in the degree of mission on-board autonomy. LOGOS is made up of a community of autonomous software agents, which exhibit autonomic behavior and cooperate to perform the functions that in the past have been performed by human operators using traditional software tools such as orbit generators and command sequence planners. It is designed to operate in "lights out" mode (i.e., without human intervention except in situations where problems and anomalies can no longer be dealt with by the system itself).

LOGOS comprises ten agents, some of which interface with legacy software, some of which perform services for the other agents in the community, and some of which interface with an analyst or operator. The agents perform the functions that in a conventional approach would be performed by human operators using traditional software tools such as orbit generators and command sequence planners.

The agents include:



```
channel analystmessage, databaserequest, message, messagenotrecognized, pagernur
UserInterface = pagerrequest ! 0 -> UserInterface ;
Database = databaserequest ? x -> pagernumber ! 0 -> Database ;
Pager = pagerrequest ? x -> databaserequest ! 0 -> Pager [] pagernumber ? x -> analystm
Analyst = analystmessage ? x -> Analyst [] stoppaggingmessage ? x -> Analyst ;
Sender = message ! 0 -> Sender [] messagenotrecognized ? x -> Sender ;
System = Analyst [| {| |} |] Database [| {| |} |] Pager [| {| |} |] Sender [| {| |} |] UserInterface ;
```

Figure 5. CSP model

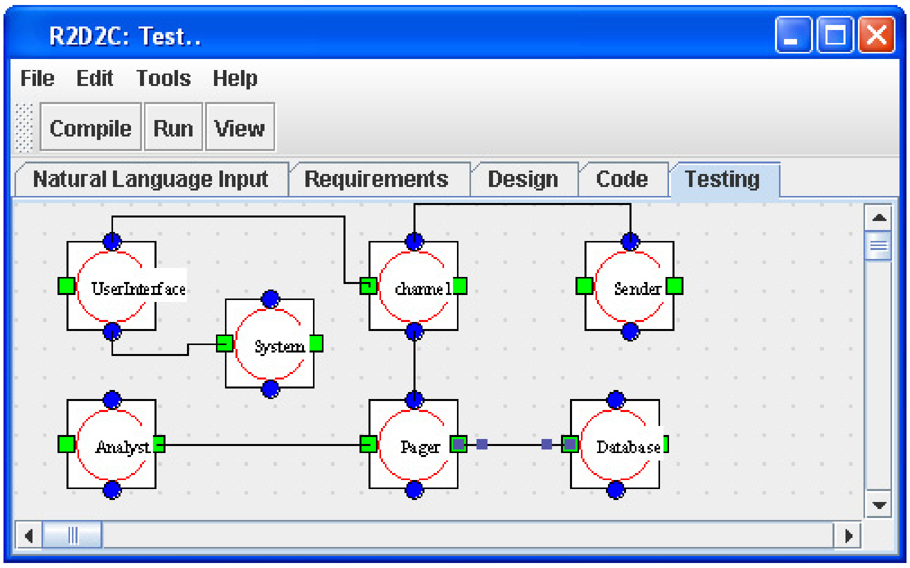


Figure 6. Graphical representation of a system

User Interface Agent: acts as an interface between the analyst and the agent community.

Spacecraft Monitoring Agent: interfaces with the spacecraft, receives the telemetry, and sends it to the proper agent.

Fault Resolution Agent: contains an expert system that can automatically fix anomalies and learn from the analyst how to resolve new anomalies.

Database Interface Agent: interfaces with a database for discrete data items (e.g., user names and passwords, pager numbers, etc.),

Trending Agent: analyzes spacecraft telemetry for trend information and sends the telemetry to an archival database for long term storage.

Pager Interface Agent: interfaces with the paging system so that an analyst can be paged in the event of an anomaly or other fault that can not be handled by LOGOS,

Visualization Interface Agent: interfaces with the data visualization system for visualizing telemetry data for the analyst.

GenSAA Interface Agent: interfaces with the spacecraft scheduling system to obtain times for subsequent spacecraft passes.

Archive Interface Agent: logs all agent messages for debugging purposes.

Spacecraft Monitoring Agent: is a manager agent that all agents register with and from which an agent may obtain addresses of other agents with which it may need to communicate.

Each agent can communicate with any other agent in the community, though not all agents need to communicate with each other. When the agents start up, they all register their capabilities with the manager agent and then request pointers to other agents that can perform needed services. When an agent registers with the requested capability, the manager agent sends the requesting agent the address of the agent with the capability. The two agents then engage in a handshake process by which the servicing agent obtains the requesting agent's address and the requesting agent verifies that the servicing agent can perform the needed service.

For more details on LOGOS and its autonomic properties, see (Rouff et al., 2004; Truszkowski et al., 2006; Rash et al., 2005)..

4.2. LOGOS in R2D2C

Although a relatively small system, the entire LOGOS system is too extensive to illustrate in its entirety in this paper. Instead, we will illustrate the application of the R2D2C approach and the operation of the prototype by considering parts of LOGOS. We will consider example agents from the system, namely the Spacecraft Monitoring Agent, the Archive Interface Agent, and the Pager Interface Agent, and illustrate their mapping from natural language descriptions through to simple Java implementations.

A trivial example will, to begin, illustrate how scenarios map to CSP. Suppose we have the following as part of one of the scenarios for the system:

> if the Spacecraft Monitoring Agent receives a “fault” advisory from the spacecraft, the agent sends the fault to the Fault Resolution Agent
> OR
> if the Spacecraft Monitoring Agent receives engineering data from the spacecraft, the agent sends the data to the Trending Agent

That part of the scenario could be mapped to structured text as:

> inSMA?fault from Spacecraft
> then outSMA!fault to FIRE
> else
> inengSMA?data from Spacecraft
> then outengSMA!data to TREND

The laws of concurrency would allow us to derive the traces as:

$$\begin{gathered} tSMA \supseteq \{\langle\rangle, \langle inSMA,\ fault\rangle, \\ \langle inSMA,\ fault,\ outSMA,\ fault\rangle\} \bigcup \\ \{\langle\rangle, \langle inengSMA,\ data\rangle, \\ \langle inengSMA,\ data,\ outSMA,\ data\rangle\} \end{gathered}$$

From the traces, we can infer an equivalent CSP process specification as:

$$\begin{aligned} SMA = \ & inSMA?fault \rightarrow (outSMA!fault \rightarrow SKIP) \\ & \mid (inengSMA?data \rightarrow outengSMA!data \rightarrow SKIP) \end{aligned}$$

Let us now consider a slightly larger example, the LOGOS Archive Interface Agent (AIFA). The purpose of the AIFA is to provide access to an archive database. The AIFA is different from the Database Interface Agent in that the archive is for storage of large amounts of telemetry data, while the database is primarily responsible for storage of smaller items to which the agents will need fast access, such as anomalies and user names. It communicates with other agents through separate input and output channels.

if the Archive Interface Agent receives a request for telemetry data, the Archive Interface Agent sends a request to the responsible agent, receives telemetry data, and sends a response back to the sender of the request
OR
if the Archive Interface Agent receives a request for mnemonic data, the Archive Interface Agent sends a request to the responsible agent, receives mnemonic data, and sends a response back to the sender of the request
OR
if the Archive Interface Agent receives a request for inserting data, the Archive Interface Agent sends a request to the archive for storage
OR
if the Archive Interface Agent receives another kind of message, reply to the sender that the message was not recognized

---

```
$AIFA\_BUS = aifa.Iin?msg \rightarrow$
  $case$
        $AIFA\_REQUEST\_TELEMETRY_{msg,telem\_name}$
              $if\ msg = (REQUEST, RETURN\_DATA, telem\_name)$

        $AIFA\_REQUEST\_MENMONIC_{msg,mnem\_name}$
              $if\ msg = (REQUEST, RETURN\_DATA, mnem\_name)$

        $aifa.Eout!telemetry \rightarrow aifa.Ein?result$
                    $\rightarrow IAFA\_BUS$
              $if\ msg = (INFORM, INSERT\_DATA, telemetry)$

        $aifa.Iout!(head(msg), UNRECOGNIZED)$
                    $\rightarrow IAFA\_BUS$
              $otherwise$
```

Figure 7. **Partial CSP description of the Archive Interface Agent.**

The above scenarios would then be translated into CSP. Figure 7 shows a partial CSP description of the AIFA. This specification states that the process AIFA_BUS receives a message on its "*Iin*" channel and stores it in a variable called "*msg*". Depending on the contents of the message, one of four different processes is executed.

The first two processes in the case statement are requests for data, the third is a request to store data, and the fourth is an error message for a malformed message. The one request for insertion is processed so that the data is sent to the archive and the agent waits for a confirmation before proceeding. There is no timeout when waiting for the archive, so the agent can deadlock; if the archive returns an error message, that message is ignored.

Figure 8 shows the specification of the two processes that request data from the archive (see the case statement in Figure 7).

$$\begin{aligned}
&AIFA\_REQUEST\_TELEMETRY_{msg,tel\_name}\\
&\qquad = aifa.Eout!tel\_name\\
&\qquad \rightarrow aifa.Ein?result\\
&\qquad \rightarrow aifa.Iout!(agent(msg), msg\_id(msg), tel\_name, tel(result))\\
&\qquad \rightarrow AIFA\_BUS
\end{aligned}$$

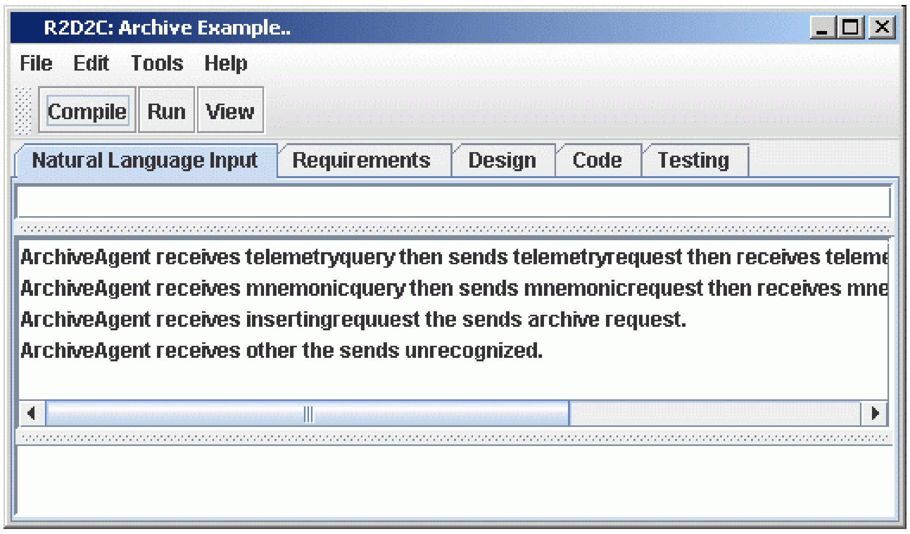
$$\begin{aligned}
&AIFA\_REQUEST\_MENMONIC_{msg,mnem\_name}\\
&\qquad = aifa.Eout!mnem\_name\\
&\qquad \rightarrow aifa.Ein?result\\
&\qquad \rightarrow aifa.Iout!(agent(msg), msg\_id(msg), mnem\_name, mnem(result))\\
&\qquad \rightarrow AIFA\_BUS
\end{aligned}$$

Figure 8. **Partial CSP description of the Archive Interface Agent.**

The requests for data are similar to requests for storing of data, except the results are sent back to the requesting agent. In both cases the request is sent over the *aifa.Eout* channel to the archive and the agent then waits for the archive to answer over the same channel (again, with no timeout, this can lead to a deadlock). Once the response from the archive is received over the *afia.Ein* channel, the result is sent back to the requesting agent over the *afia.Iout* channel. The name of the requesting agent and the original message's id are extracted from the message representing the request for data.

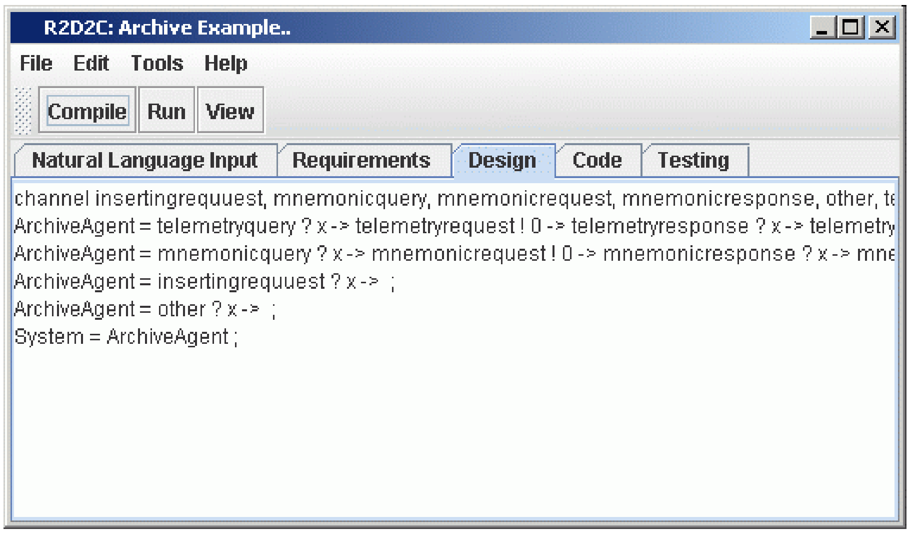
The corresponding R2D2C tool snapshots are shown in Figures 9 and 10.

A similar approach for the Pager Interface Agent will produce



```
ArchiveAgent receives telemetryquery then sends telemetryrequest then receives teleme
ArchiveAgent receives mnemonicquery then sends mnemonicrequest then receives mne
ArchiveAgent receives insertingrequuest the sends archive request.
ArchiveAgent receives other the sends unrecognized.
```

Figure 9. Archive Interface Agent Input requirements



```
channel insertingrequuest, mnemonicquery, mnemonicrequest, mnemonicresponse, other, te
ArchiveAgent = telemetryquery ? x -> telemetryrequest ! 0 -> telemetryresponse ? x -> telemetry
ArchiveAgent = mnemonicquery ? x -> mnemonicrequest ! 0 -> mnemonicresponse ? x -> mne
ArchiveAgent = insertingrequuest ? x ->  ;
ArchiveAgent = other ? x ->  ;
System = ArchiveAgent ;
```

Figure 10. Archive Interface Agent CSP model

if the Pager Interface Agent receives a request from the User Interface Agent, the Pager Interface Agent sends a request to the Database Interface Agent for an analyst's pager information and puts the message in a list of requests to the Database Interface Agent
OR
if the Pager Interface Agent receives a pager number from the Database Interface Agent, then the Pager Interface Agent

removes the message from the paging queue and sends a message to the analyst's pager and adds the analyst to the list of paged people

OR

if the Pager Interface Agent receives a message from the User Interface Agent to stop paging a particular analyst, the Pager Interface Agent sends a stop-paging command to the analyst's pager and removes the analyst from the paged list

OR

if the Pager Interface Agent receives another kind of message, reply to the sender that the message was not recognized

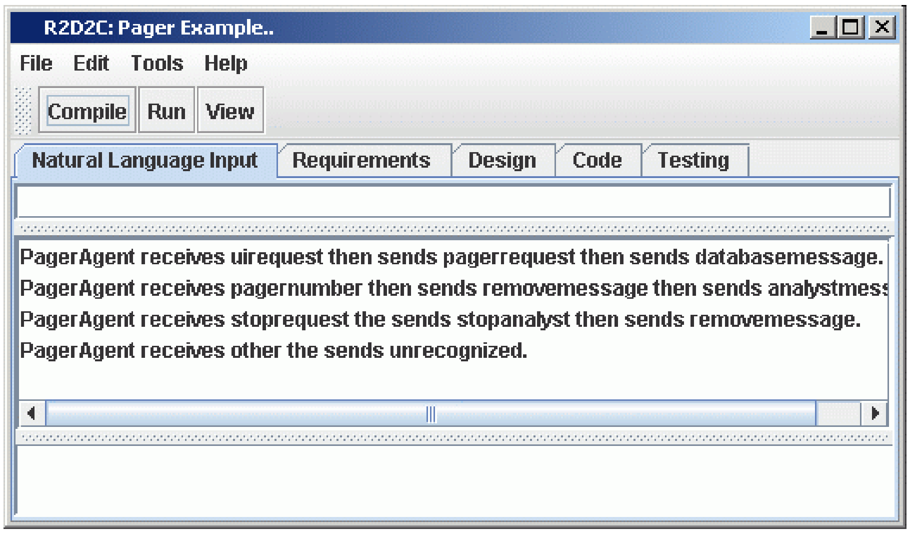
The corresponding CSP is shown in Figure 11 and the corresponding R2D2C tool snapshots are shown in Figures 12 and 13.

```
$PAGER\_BUS_{db\_waiting,paged} = pager.Iin?msg \rightarrow$
  $case$
      $GET\_USER\_INFO_{db\_waiting,paged,pagee,text}$
          $if\ msg = (START\_PAGING, specialist, text)$

      $BEGIN\_PAGING_{db\_waiting,paged,in\_reply\_to\_id(msg),pager\_num}$
          $if\ msg = (RETURN\_DATA.pager\_num)$

      $STOP\_CONTACT_{db\_waiting,paged,pagee}$
          $if\ msg = (STOP\_PAGING, pagee)$

      $pager.Iout!(head(msg), UNRECOGNIZED)$
                  $\rightarrow PAGER\_BUS_{db\_waiting,paged}$
          $otherwise$
```

Figure 11. **Partial CSP description of the pager agent.**

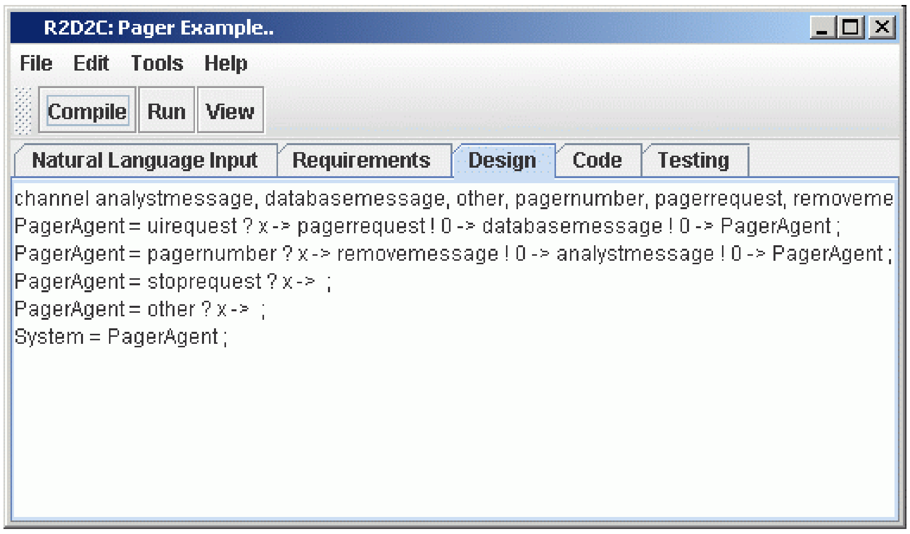
4.3. Results

A formal specification of LOGOS in CSP had previously been undertaken by hand (Rouff et al., 2000). This afforded numerous insights, highlighting over 80 errors and anomalies in the requirements of a relatively small system (LOGOS is based, essentially, on ten interacting



```
PagerAgent receives uirequest then sends pagerrequest then sends databasemessage.
PagerAgent receives pagernumber then sends removemessage then sends analystmess
PagerAgent receives stoprequest the sends stopanalyst then sends removemessage.
PagerAgent receives other the sends unrecognized.
```

Figure 12. Pager Interface Agent Input requirements



```
channel analystmessage, databasemessage, other, pagernumber, pagerrequest, removeme
PagerAgent = uirequest ? x -> pagerrequest ! 0 -> databasemessage ! 0 -> PagerAgent ;
PagerAgent = pagernumber ? x -> removemessage ! 0 -> analystmessage ! 0 -> PagerAgent ;
PagerAgent = stoprequest ? x ->  ;
PagerAgent = other ? x ->  ;
System = PagerAgent ;
```

Figure 13. Pager Interface Agent CSP model

agents). While many of these were minor oversights that would have caused inconveniences, others were more significant.

A great advantage of using an example for which we already have a formal specification is that we can compare the system derived by our prototype tool with the manually derived formal specification.

Our prototype tool was able to uncover all of the errors and anomalies we found with our manual specification. We were surprised when we first ran it to find that it halted within seconds, having found yet another error that had been introduced into the requirements (due to a typographical error) when changes were made following the original manual formal specification. The prototype tool can cope with the LOGOS requirements, generating a design and a Java implementation in a matter of minutes, whereas manual specification had taken several days and code generation by hand took several weeks.

A number of additional prospective R2D2C applications have been identified, implementations of which could be derived both manually and using R2D2C—from which could be collected measurements that would support statistical comparisons of the manual and automated R2D2C appproaches. Such measurements could include time to implement, lines of code, and execution speed. Information of this nature concerning prototyped applications indicates a definite advantage for automated generation of implementations for at least the limited domains addressed thus far.

More fundamental is the question of the range of applicability of the automated approach defined by the R2D2C method, as well as the question of the efficiency and comprehensiveness of automated verification capabilities. Further case studies and prototyping over time will provide answers to these questions.

5. Future Applications

The prototype tool described in this paper is designed to support a NASA patent-pending method for Requirements-Based Programming (RBP). The uniqueness of the method is not in supporting RBP, but in supporting it with a development process that is mathematically tractable over the entire development process. This fully formal development offers levels of assurance and confidence significantly higher than traditionally available.

The method is not limited to producing executable code, however (Hinchey et al., 2005a). In addition to applying the approach to agent-based systems (such as LOGOS) as described in this paper, and to Wireless Sensor Networks (WSNs) (Hinchey et al., 2005b), we are currently examining applications of the approach to the verification of expert systems and robotic applications.

6. Conclusions

The difficulty of developing many autonomous and autonomic applications is explained by their inherent complexity. Often, required autonomous behavior results in emergent, unexplained behavior that could not, reasonably, have been foreseen. The need to exhibit autonomic behavior often compounds the situation, giving rise to necessary self-managing behavior that could not reasonably be expected to be the subject of even the most exhaustive testing plans.

Only with fully formal underpinnings for the development process can we be assured of correctness (Bauer, 1980). Formal development processes will become more and more important in future autonomic computing systems, and the continued success of the Autonomic Computing initiative is predicated on the ability to develop complex systems that both exhibit autonomic self-managing behaviors *and* operate correctly (with respect to their requirements).

The experience related in this paper leads us to be confident that such tools will offer greater levels of assurance in other domains, and enhance both the quality and performance of future autonomous and autonomic systems.

Acknowledgements

This paper is a revised and extended version of (Rash et al., 2005) presented at IEEE EASe 2005. Part of this work was supported by the NASA Goddard Space Flight Center Technology Transfer Office. Denis Gračanin was supported by an ASEE/NASA Summer Faculty Fellowship hosted at the NASA Software Engineering Laboratory (Code 581), NASA Goddard Space Flight Center. John Erickson was supported by the NASA Student Internship Program and by the Information Systems Division (Code 580) at NASA Goddard Space Flight Center.

References

Bauer, F. L.: 1980, 'A trend for the next ten years of software engineering'. In: H. Freeman and P. M. Lewis (eds.): *Software Engineering*. Academic Press, pp. 1–23.

Bowen, J. P. and M. G. Hinchey: 1995, 'Seven More Myths of Formal Methods'. *IEEE Software* **12(4)**, 34–41.

Gosling, J., B. Joy, G. Steele, and G. Bracha: 2000, *Java^{TM} Language Specification*. Boston: Addison Wesley, second edition.

Harel, D.: 2001, 'From Play-In Scenarios To Code: An Achievable Dream'. *IEEE Computer* **34(1)**, 53–60.

Harel, D.: 2004, 'Comments made during presentation at "Formal Approaches to Complex Software Systems" panel session'. *ISoLA-04 First International Conference on Leveraging Applications of Formal Methods*.

Hinchey, M. G. and J. P. Bowen (eds.): 1999, *Industrial-Strength Formal Methods in Practice*, FACIT Series. London, UK: Springer-Verlag.

Hinchey, M. G. and S. A. Jarvis: 1995, *Concurrent Systems: Formal Development in CSP*, International Series in Software Engineering. London, UK: McGraw-Hill International.

Hinchey, M. G., J. L. Rash, and C. A. Rouff: 2004, 'Requirements to Design to Code: Towards a Fully Formal Approach to Automatic Code Generation'. Technical Report TM-2005-212774, NASA Goddard Space Flight Center, Greenbelt, MD, USA.

Hinchey, M. G., J. L. Rash, and C. A. Rouff: 2005a, 'A Formal Approach to Requirements-Based Programming'. In: *Proc. IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005).* IEEE Computer Society Press, Los Alamitos, Calif.

Hinchey, M. G., J. L. Rash, and C. A. Rouff: 2005b, 'Towards an Automated Development Methodology for Dependable Systems with Application to Sensor Networks'. In: *Proc. IEEE Workshop on Information Assurance in Wireless Sensor Networks (WSNIA 2005), Proc. International Performance Computing and Communications Conference (IPCCC-05).* Phoenix, Arizona, IEEE Computer Society Press, Los Alamitos, Calif.

Hoare, C. A. R.: 1978, 'Communicating Sequential Processes'. *Communications of the ACM* **21(8)**, 666–677.

Hoare, C. A. R.: 1985, *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science. Englewood Cliffs, NJ: Prentice Hall International.

Kleppe, A., J. Warmer, and W. Bast: 2003, *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston: Addison-Wesley.

Lea, D.: 2000, *Concurrent Programming in Java^{TM}: Design Principles and Patterns*, The Java^{TM} Series. Reading, Massachusetts: Addison-Wesley Professional, second edition.

Parr, T. J. and R. W. Quong: 1995, 'ANTLR: A Predicated-LL*k* Parser Generator'. *Software Practice and Experience* **25**(7), 789–810.

Rash, J. L., M. G. Hinchey, C. A. Rouff, D. Gračanin, and J. D. Erickson: 2005, 'Experiences with a Requirements-Based Programming Approach to the Development of a NASA Autonomous Ground Control System'. In: *Proc. IEEE Workshop on Engineering of Autonomic Systems (EASe 2005) held at the IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005).* IEEE Computer Society Press, Los Alamitos, Calif.

Rouff, C. A., J. L. Rash, and M. G. Hinchey: 2000, 'Experience Using Formal Methods for Specifying a Multi-Agent System'. In: *Proc. Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000).* Tokyo, Japan, IEEE Computer Society Press, Los Alamitos, Calif.

Rouff, C. A., W. F. Truszkowski, M. G. Hinchey, and J. L. Rash: 2004, 'Verification of Emergent Behaviors in Swarm Based Systems'. In: *Proc. 11th IEEE International Conference on Engineering Computer-Based Systems (ECBS), Workshop on Engineering Autonomic Systems (EASe).* Brno, Czech Republic, pp. 443–448, IEEE Computer Society Press, Los Alamitos, Calif.

Schneider, S., J. Davies, D. M. Jackson, G. M. Reed, J. Reed, and A. W. Roscoe: 1991, 'Timed CSP: Theory and Practice'. In: *Proc. REX, Real-Time: Theory in Practice Workshop*, Vol. 600 of *LNCS*. pp. 640–675, Springer-Verlag.

Smaragdakis, Y., S. S. Huang, and D. Zook: 2004, 'Program generators and the tools to make them'. In: *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* pp. 92–100, ACM Press.

Sterritt, R. and M. G. Hinchey: 2005, 'Why Computer Based Systems *Should* be Autonomic'. In: *Proc. 12th IEEE International Conference on Engineering of Computer Based Systems (ECBS 2005).* Greenbelt, MD, pp. 406–414.

Truszkowski, W. F., M. G. Hinchey, J. L. Rash, and C. A. Rouff: 2006, 'Autonomous and Autonomic Systems: A Paradigm for Future Space Exploration Missions'. *IEEE Transactions on Systems, Man and Cybernetics, Part C (to appear).*

Truszkowski, W. F., J. L. Rash, C. A. Rouff, and M. G. Hinchey: 2004, 'Some Autonomic Properties of Two Legacy Multi-Agent Systems — LOGOS and ACT'. In: *Proc. 11th IEEE International Conference on Engineering Computer-Based Systems (ECBS), Workshop on Engineering Autonomic Systems (EASe).* Brno, Czech Republic, pp. 490–498, IEEE Computer Society Press, Los Alamitos, Calif.

Walrath, K., M. Campione, A. Huml, and S. Zakhour: 2004, *JFC Swing Tutorial, The: A Guide to Constructing GUIs.* Boston: Addison Wesley, second edition.

Welch, P. H., J. R. Aldous, and J. Foster: 2002, 'CSP Networking for Java (JCSP.net)'. In: *Proceedings of the Global and Collaborative Computing Workshop (ICCS 2002)*, Vol. 2330 of *Lecture Notes in Computer Science.* pp. 695–708, Springer-Verlag.

Biographies

James L. Rash received the MA in Mathematics from the University of Texas at Austin, USA, and the BA degree in Mathematics and Physics from the University of Texas at Austin, USA. He leads formal methods research and development in the Advanced Architectures and Automation Branch at the NASA Goddard Space Flight Center, where his other major responsibilities include managing the Operating Missions as Nodes on the Internet (OMNI) Project. He has authored/co-authored move than 30 technical papers and articles, co-edited three books, and edited eight journal special issues, and has been an organizer of more than 15 conferences and workshops on artificial intelligence, formal methods, and Internet technologies for space missions. His research interests include formal methods and agent-based technologies.

Michael G. Hinchey received the PhD in Computer Science from University of Cambridge, UK, the MSc degree in Computation from University of Oxford, UK, and the BSc degree in Computer Science from University of Limerick, Ireland. He is currently Director of the NASA Software Engineering Laboratory, located at Goddard Space Flight Center. Prior to joining the US Government, he held academic positions at the level of Full Professor in the USA, UK, Ireland, Sweden and Australia. He is the author of more than 200 technical papers, and 15 books. His current research interests are in the areas of formal methods, system correctness, and agent based technologies. Dr Hinchey is a Senior Member of the IEEE, a Fellow of the IEE, the Institute of Mathematics and Its Applications, the Institute of Engineers of Australia, and the British Computer Society. He is a Chartered Engineer, Chartered Professional Engineer, Chartered Information Technology Professional and Chartered Mathematician. He is currently Chair of the IEEE Technical Committee on Complexity in Computing, and is the IEEE Computer Societys voting representative to IFIP TC1, of which he has been elected Chair for 2006 to 2008.

Christopher Rouff received the PhD in Computer Science from the University of Southern California, a MS in Computer Science from University of California, Davis and a BA in Mathematics/Computer Science from California State University, Fresno. He is currently a senior scientist in the Advanced Concepts Business Unit at Science Applications International Corporation and is conducting research and development on multi-agent systems, verification of intelligent systems and collaborative robotics for NASA and DARPA. Previously he was with NASA Goddard for nine years where he researched and prototyped cooperative multi-agent systems for ground and spaceflight applications and led a number of software research and development projects. Dr Rouff has over seventy publications and twenty-five years of experience in software engineering and intelligent systems.

Denis Gracanin was born in Rijeka, Croatia, on May 12, 1963. He received the BS and MS degrees in electrical engineering from the University of Zagreb, Croatia, in 1985 and 1988, respectively, and the MS and PhD degrees in computer science from the University of Louisiana at Lafayette in 1992 and 1994, respectively. He was a Research Scientist in the A-CIM Center, University of Louisiana at Lafayette, from January 1994 to August 1999, and an adjunct Assistant Professor in the Center for Advanced Computer Studies, University of Louisiana at Lafayette, from August 1997 to August 1999. Since August 1999, he has been an Assistant Professor in the Computer Science Department, Virginia Polytechnic Institute & State University. In 2004 he received NASA summer faculty fellowship and in 2005 he received the ONR/ASEE summer senior faculty fellowship. His current research interests include distributed virtual environments, distributed simulations, and sensor networks. Dr Gracanins professional memberships include the ACM, AAAI, APS, SCS, and SIAM. He is also a Professional Engineer in electrical engineering, Louisiana Professional Engineering and Land Surveying Board. In 1991, he received a Fulbright scholarship for studies at the University of Louisiana at Lafayette.

John Erickson is a PhD student in Computer Sciences at The University of Texas at Austin. He is interested in formal verification using automatic theorem proving, and is currently investigating ways to improve the ACL2 theorem prover. His advisor is J Moore and he plans to graduate in December of 2006.

Address for Offprints: Kluwer Prepress Department
P.O. Box 990
3300 AZ Dordrecht
The Netherlands